Avalanche: An Event Dispatcher for SNO+

(as in, a lot of sno(+), very fast)

Andy Mastbaum*

The University of Pennsylvania

January 31, 2012

1 Introduction

The "dispatcher" is the server software responsible for providing a stream of built events to various client software. Currently-planned client software includes the event viewer and detector monitoring tools. The dispatcher requires:

High Throughput The dispatcher must be able to keep up with the maximum data rate of 450 Mbps, simultaneously to all clients.

Scalability The dispatcher should support a large number of concurrent client connections without impact on performance.

Compatibility The dispatched data stream must be reasonably language-agnostic, since the language of the client software is unknown.

An obvious platform choice would be ROOT's network I/O classes (`TPSocket`, etc.). However, the next-generation socket library ZeroMQ[1] provides a superior alternative. ZeroMQ is naturally asynchronous, natively supports fan-out, and implementations are typically faster than UNIX sockets. ZeroMQ's fan-out allows clients to "subscribe" to a stream, optionally applying a filter (e.g. only events, no run headers). Bindings exist for all major languages.

2 Data Format

It has been decided that the "packed" events coming from the event builder will be in a ROOT format, with a structure developed by G. D. Orebi Gann and J. Kaspar. The packed format preserves the "raw" data structures from the detector front-end: for example, charge and time data is stored in 96-bit PMT bundles. This is in contrast the 'full' or 'unpacked' data structure used in RAT, where values are broken into separate, histogrammable fields.

The packed data is essentially a stream of `TObjects`, which may be event data, event-level headers, or run-level headers. For convenience of storage these are written to a `TTree`. Since `TTree`s are homogeneous lists, various data types must inherit from the same generic superclass to be stored in the tree. The data type must also be stored, so that objects can be properly re-cast when read back. The packed format `TTree` stores `RAT::DS::PackedRec` objects, which in turn contain a single `GenericRec` and an integer `RecordType`. All data types (events, headers) inherit from a class `GenericRec`, which is empty and exists only to support this polymorphism.

A full listing of packed format classes and their members is given in the appendix.

*mastbaum@hep.upenn.edu

[1] http://www.zeromq.org

3 Network Layer

The avalanche server broadcasts `PackedRec` objects, serialized using a ROOT `TBufferFile`. On the client side, software must create a `TBufferFile` using the packet data, read the `PackedRec` object out of it, and cast the `PackedRec`'s `Rec` object to the correct type based on the value of `RecordType`. C++ and Python libraries to perform this deserialization are included in with avalanche, and examples in both languages are given in the `examples/` directory.

4 CouchDB Integration

Avalanche clients may also connect to a Couch database and receive headers as they are pushed in by the DAQ. These notifications are retrieved via the CouchDB changes feed[2] and the `couchdb-python` Python package[3]. Headers from CouchDB are interleaved with the rest of the data stream and indistinguishable from those received from an avalanche server.

5 Dispatched Data in RAT

Using the avalanche libraries, an input producer and output processor have been developed that allows RAT to interact with dispatched data. Simulated events may be dispatched on the network, simulating the real data flow chain, and RAT can read in dispatched events, which may be used for online processing or writing monitoring tools as RAT processors.

The code to interact with dispatched data streams was committed to SVN in r652 (git revision f5387a0b495ca4b719a3a7e41487920b40f463bb). Documentation can be found in SNO+-doc-1300.

6 Dispatching Detector Events

A C++ library with a very simple API is provided in the `lib/cpp` directory.

A server consists of a single `avalanche::server` object, constructed with a given socket address. The `avalanche::server` has one method, `sendObject`, which dispatches a ROOT `TObject`. For example, to send a histogram `TH1F* h1`:

```
avalanche::server* serv = new avalanche::server("tcp://localhost:5024");
serv->sendObject(h1);
```

Any number of clients may be subscribed to the stream on port 5024.

7 Writing Dispatcher Clients

Client software connects to a stream and/or database, receives and reconstructs ROOT objects, and does something with these objects. Libraries for C++ and Python are provided in `lib/` that manage the first two steps.

7.1 C++

`libavalanche` provides a dispatcher client `avalanche::client`.

Consider the following example:

```
#include <iostream>
#include <TH1F.h>
#include <RAT/DS/PackedEvent.hh>
```

[2]http://wiki.apache.org/couchdb/HTTP_database_API#Changes

[3]http://code.google.com/p/couchdb-python/

```
#include <avalanche.hpp>

int main(int argc, char* argv[]) {
  // create a client, listening for objects on port 5024
  avalanche::client client("tcp://localhost:5024");

  // you can listen to an unlimited number of stream, using addServer
  client.addServer("tcp://localhost:5025");

  // receive RAT::DS::PackedRec objects
  while (1) {
    RAT::DS::PackedRec* rec = (RAT::DS::PackedRec*) client.recvObject(RAT::DS::PackedRec::Class());
    if (rec)
      std::cout << "Received PackedRec of type " << rec->RecordType << std::endl;
    else
      std::cout << "Error deserializing message data" << std::endl;
    delete rec;
  }

  return 0;
}
```

First, we set up the client by giving it a server address to connect to. In this example, we also connect to one more server. You may connect to an unlimited number of streams; the packets received are interleaved, as if they arrived from one source. Next, we receive ROOT objects with `recvObject`. The class must be specified for the deserialization step. Now we are ready to operate on the data. In this example we just print the record type; a real client would at this point cast the member `Rec` object to the correct type, extract data, and perform some operations.

Note that `recvObject` is blocking by default (it will wait until data is received). To use non-blocking I/O, provide the `ZMQ_NOBLOCK` flag as the second argument. When used in this mode, `recvObject` will return `NULL` when no data is available.

7.2 Python

The `avalanche` Python package in `lib/python` provides a dispatcher client `avalanche.Client`. To install this package on your system, run `$ python setup.py install`.

Consider the following example:

```
import avalanche
from rat import ROOT

if __name__ == '__main__':
    # create client listening to localhost port 5024
    cli = avalanche.Client('tcp://localhost:5024')

    # you can listen to an unlimited number of stream, using add_server
    cli.ad_server('tcp://localhost:5025');

    # receive RAT::DS::PackedRec objects
    while True:
        rec = cli.recv_object(ROOT.RAT.DS.PackedRec.Class())

        if rec:
            print 'Received PackedRec of type', rec.RecordType
        else:
```

```
        print 'Error deserializing message data'
```

First, we set up the client by giving it a server address to connect to. In this example, we also connect to one more server. You may connect to an unlimited number of streams; the packets received are interleaved, as if they arrived from one source. Next, we receive ROOT objects with `recv_object`. The class must be specified for the deserialization step. Now we are ready to operate on the data. In this example we just print the record type; a real client would at this point cast the member `Rec` object to the correct type, extract data, and perform some operations.

Note that in Python, there is no need to cast the object returned by `recv_object`, as this is taken care of by PyROOT.

As above, `recv_object` is blocking by default (waits until data is available). To use non-blocking I/O, provide the `flags=zmq.NOBLOCK` argument. When used in this mode, `recv_object` will return `None` when no data is available.

7.3 Other Languages

Many other languages are supported by ZeroMQ, and at least Ruby has supported ROOT bindings. Implementations will be similar to those given in C++ and Python. For examples of ZeroMQ in your language, see https://github.com/imatix/zguide/tree/master/examples.

A client must 'subscribe' to a ZeroMQ TCP 'publish' socket and deserialize ROOT `TObjects` using a `TBufferFile`. The former is achieved with the ZeroMQ API: set up a context and a subscriber socket, and connect that socket to one or more server addresses. The latter is simple in principle – create a `TBufferFile`, set the buffer to the packet contents, and read out the appropriate class. This may be complicated (as it is in Python) by the language's hadling of string termination; care must be taken to ensure that the buffer is not treated as a string and truncated at the first null byte. For guidance, see the C++ and Python client library source code in `lib`.

CouchDB bindings also exist for many languages, but none are essential since Couch communicates using JSON strings sent over HTTP. To interact with a database, code needs to make HTTP queries and convert JSON strings into native objects. All major languages have such facilities. If no library exists for your language, details on the CouchDB changes API are available at http://wiki.apache.org/couchdb/HTTP_database_API#Changes.

8 Appendix: Packed Data Model

This is a complete listing of all packed format classes, taken from RAT's PackedEvent.hh.

8.1 class GenericRec : public TObject

(empty)

8.2 class PackedRec : public TObject

- UInt_t RecordType
- GenericRec *Rec

8.2.1 Record Types

0. Empty
1. Detector event
2. RHDR
3. CAAC
4. CAST
5. TRIG
6. EPED

8.3 class PackedEvent : public GenericRec

- UInt_t MTCInfo[kNheaders] (6 words for the event header from the MTC)
- UInt_t RunID
- UInt_t SubRunID
- UInt_t NHits
- UInt_t EVOrder
- ULong64_t RunMask
- char PackVer
- char MCFlag
- char DataType
- char ClockStat10
- std::vector<PMTBundle> PMTBundles

8.4 class PMTBundle

- UInt_t Word[3]

8.5 class EPED : public GenericRec

- UInt_t GTDelayCoarse
- UInt_t GTDelayFine
- UInt_t QPedAmp
- UInt_t QPedWidth
- UInt_t PatternID
- UInt_t CalType
- UInt_t EventID (GTID of first events in this bank validity)
- UInt_t RunID (doublecheck on the run)

8.6 class TRIG : public GenericRec

- UInt_t TrigMask
- UShort_t Threshold[10]
- UShort_t TrigZeroOffset[10]
- UInt_t PulserRate
- UInt_t MTC_CSR
- UInt_t LockoutWidth
- UInt_t PrescaleFreq
- UInt_t EventID (GTID of first events in this banks validity)
- UInt_t RunID (doublecheck on the run)

8.6.1 Array Indices

Arrays correspond to:

0. N100Lo
1. N100Med
2. N100Hi
3. N20
4. N20LB
5. ESUMLo
6. ESUMHi
7. OWLn
8. OWLELo
9. OWLEHi

8.7 class RHDR : public GenericRec

- UInt_t Date
- UInt_t Time
- char DAQVer
- UInt_t CalibTrialID
- UInt_t SrcMask
- UInt_t RunMask
- UInt_t CrateMask
- UInt_t FirstEventID
- UInt_t ValidEventID
- UInt_t RunID (doublecheck on the run)

8.8 class CAST : public GenericRec

- UShort_t SourceID
- UShort_t SourceStat
- UShort_t NRopes
- float ManipPos[3]
- float ManipDest[3]
- float SrcPosUncert1
- float SrcPosUncert2[3]
- float LBallOrient
- std::vector<int> RopeID
- std::vector<float> RopeLen
- std::vector<float> RopeTargLen
- std::vector<float> RopeVel
- std::vector<float> RopeTens
- std::vector<float> RopeErr

8.9 class CAAC : public GenericRec

- float AVPos[3]
- float AVRoll[3] (roll, pitch and yaw)
- float AVRopeLength[7]